1 Parser

Once the actual code has been separated from proof code, the Parser then parses into an abstract syntax tree.

```
module Parser where
  import Data.Map (Map)
  import qualified Data.Map.Strict as Map
  import Lexer (lexify)
  import Happy
  import AST
  import Result
```

The NativeAST is, for now, a placeholder for whatever type is produced by the lexer/parser of the language being proven.

```
  data NativeAST = NativeASTNode

  parseCode :: String → AST
  parseCode code = transformAST NativeASTNode

  transformAST :: NativeAST → AST
  transformAST native = ID "The code" VEmpty
```

Once the code has been turned into a NativeAST, it is then transformed into the AST by the pluggable Transformer. Meanwhile, the proof code must also be converted into the definitions used to prove the program. These are represented by the same AST as the code, but this transformation is handled here.

The first step in parsing the proof code is, of course, lexifying it. This step is taken on by the Lexer. After that, we move on to parsing, which uses the parser generated by Happy.

```
  parseProofs :: String → Result AST
  parseProofs proofText = parse $ lexify proofText
```

Finally, the proof and native code ASTs are combined into one, inserting the code at the `Insert` points and annotating wherever possible.

```
annotates :: AST → AST → AST
annotates a b = a
```